Introduzione a Standard ML

`print "help\n";`

Emanuele Nardi

Compilato il 11 febbraio 2021
v1.0.0

Impostazione ambiente di lavoro

Il modo più semplice per imparare Standard ML è quella di interagire tramite un Read–Eval–Print Loop (REPL). Per installarlo ed eseguirlo in ambiente Linux:

```
$ sudo apt install polyml rlwrap    # installazione
$ alias poly='rlwrap poly'          # defizione di alias
$ poly                              # esecuzione della REPL
```

Altrimenti, se usi un sistema windows, ti consiglio di utilizzare una macchina virtuale per simulare lo stesso ambiente che ti si presenterà in sede d'esame. Se proprio non puoi utilizzare una macchina virtuale allora puoi scaricare l'ultimo binario precompilato di Poly ML da github. Per eseguire il codice fornito è possibile caricare il programma nel dal terminale con il comando "`use`":

```
use "exec.sml"
```

Per interrompere la compilazione il comando da eseguire è `ctrl + c`, mentre per uscire dalla sub-shell di poly il comando è `ctrl + z`.

Fonti

Questi appunti si basano sulla playlist "Functional Programming with Standard ML" visualizzata ad oggi all'incirca un migliaio di volte, ho pensavo fosse utile averne una trascrizione come traccia e veloce riferimento. All'interno della playlist sono presenti video dei corsi online di linguaggi di programmazione della Reykjavik University e della Washington University, insegnati da Hrafn Loftsson e Dan Grossman nell'inverno e nella primavera del 2013 rispettivamente.

Per quanto riguarda il corso di Hrafn Loftsson non sono riuscito a trovare ulteriori materiali oltre ai video pubblicati sul canale RU Computer Science.

Mentre per quanto riguardo il corso di Dan Grossman è possibile fruire dei materiali direttamente dal suo sito che, seppur presentando una grafica spartana, fornisce tutto il materiale necessario a seguire il corso nel dettaglio, sono persino presenti i sorgenti dei codici che vedrete nei suoi video.

Dan Grossman ha fornito in parallelo il corso sulla piattaforma Coursera ma seppur i due corsi di sovrappongano non sono presentati nella stessa maniera (come precisato nel documento "Relation to Coursera Course").

Recensire tutto questo materiale comporterebbe un grosso investimento di tempo che sperabilmente io riuscirò ad evitarti.

Se pensi che questi appunti ti siano stati utili ti chiedo gentilmente di fare una piccola donazione.

Introduzione

La fondazione teorica della programmazione funzionale di basa sul lambda calcolo *lamda calculus*. SML (Standard ML) è un linguaggio di programmazione derivato dall'ML (Meta Language). L'ML originale era una serie di MetaLinguaggi ideati da Robin Milner (e dai suoi studenti) all'Università di Edimburgo per creare programmi che eseguissero la dimostrazione di teoremi. Questi metalinguaggi furono poi "standardizzati" per dare origine all'SML, di cui lo standard più recente risale al 1997. L'SML è un linguaggio funzionale, quindi avente la caratteristica di rendere facile ed efficiente la creazione e l'uso di funzioni specializzate (dalla voce "Standard ML" da Wikipedia in italiano).

La programmazione funzionale è un paradigma di programmazione in cui il flusso di esecuzione del programma assume la forma di una serie di valutazioni di funzioni matematiche. Il punto di forza principale di questo paradigma è la mancanza di effetti collaterali (side-effect) delle funzioni, il che comporta una più facile verifica della correttezza e della mancanza di bug del programma e la possibilità di una maggiore ottimizzazione dello stesso. La programmazione funzionale pone maggior accento sulla definizione di funzioni, rispetto ai paradigmi procedurali e imperativi, che invece prediligono la specifica di una sequenza di comandi da eseguire. In questi ultimi, i valori vengono calcolati cambiando lo stato del programma attraverso delle assegnazioni; un programma funzionale, invece, è immutabile: i valori non vengono trovati cambiando lo stato del programma, ma costruendo nuovi stati a partire dai precedenti (dalla voce "Programmazione funzionale" da Wikipedia in italiano).

Lettura consigliata

Un libro di riferimento di questo corso è "Programming in Standard ML" di cui consiglio la lettura a partire dalla seconda parte "The Core Language" a pagina 13 leggendo dal secondo all'undicesimo capitolo saltando tutti i restanti in quanto non preparano al superamento della prova d'esame; leggendo quindi da pagina 13 a pagina 102.

Caratteristiche del linguaggio

SML è un linguaggio intepretato, usa la notazione infissa ed ha le dichiarazioni di tipo (*type declarations*), usa l'inferenza di tipo (*type inference*) (ad esempio se `x` ed `y` sono di tipo `int`, allora `x+y` è di tipo `int`).

```
2 + 3 * 4;
val it = 14 : int
```

L'interprete risponde che il valore immesso "it" è di tipo intero.

```
2.0 + 3.0 * 4.0;
val it = 14.0 : int
```

Mentre in questo caso interpreta correttamente i tipi come reali.

É un linguaggio fortemente tipato (*strongly typed*) e questo permette di determinare il tipo di ogni nome ed espressione a tempo di compilazione. Al contrario di un linguaggio come il Python dove il tipo è inferito a tempo di compilazione dall'inteprete.

Può gestire le eccezioni e ha la possibilità di creare moduli per creare dati astratti (ADT).

Liste

Le liste sono una sequenza di 0 o più elementi *dello stesso tipo*, delimitate da parentesi quadre e con la virgola come separatore di elementi.

```
[1,2,3]
val it = [1,2,3] : int list

["John", "Mary"]
val it = ["John", "Mary"] : string list

[] (*empty list*)
val it = [] :'a list (*a list of some type*)
```

Non abbiamo specificato il tipo di elementi presenti all'interno della lista, ma il linguaggio inferisce autonomamente che tipo di lista sia dal tipo di elementi contenuti al suo interno. In particolare notiamo che nella dichiarazione della lista vuota SML non sa determinare il tipo della lista.

Un lista può essere sia una lista vuota, dichiarata con `[]`, oppure nella forma `a::y`, dove `a` è la testa e `y` è la coda. Ad esempio `[7]` è uguale a `7::[]`. Questa sintassi può essere utilizzata per creare liste ed estrarre i suoi elementi.

```
7::[];
val it = [7]: int list
```

In questo caso la testa è 7, mentre la coda è la lista vuota.

Operazioni sulle liste

In quanto le liste sono gli elementi di base del linguaggio, dobbiamo aver la possibilità di manipolarle tramite operazioni sulle liste:

`null(x)`	ritorna true se la lista è vuota
`hd(x)`	ritorna la testa di x
`tl(x)`	ritorna la coda di x
`a::x`	costruisce una lista con testa `a` e coda `x`

Tabella 1

```
hd([1,2,3]);
val it = [1] : int list

tl([1,2,3]);
val it = [2,3] : int list
```

Nota che la testa di una lista è sempre un elemento (che potrebbe essere una lista), mentre la coda è una lista di elementi (che in questo caso è una lista che contiene gli elementi 2 e 3). Possiamo quindi sfruttare questo meccanismo per inserire elementi nella testa della lista e costruirne una nuova.

```
1::[2,3];
val it = [1,2,3] : int list
```

Il numero 1 costituisce la testa della lista, mentre la coda è una lista di 2 elementi.

Funzioni

Le funzioni in ML si definiscono con la seguente sintassi

```
fun <name> <formal-parameters> = <body>
```

Ad esempio

```
fun succ n = n + 1;
val succ = fn : int -> int
```

L'interprete mi sta dicendo che "succ" è una funzione che mappa un integer in un integer. La prima "n" costituisce il paramentro formale, mentre "n+1" il paramentro attuale.

Ancora una volta che non ho specificato il tipo del parametro della funzione ma riesce correttamente ad inferire il tipo.

> Come riesce ML ad inferire che la funzione tratta dei numeri interi?

Nota che applico ad una variabile una funzione di somma con un intero, in quanto il risultato della somma sarà un valore intero, allora la funzione tornerà anchessa un intero.

Le funzioni si possono richiamare sia scrivendo f(x) sia scrivendo f x. Quindi se voglio richiamare la funzione appena dichiarata posso sia scrivendo "`succ(3);`" che "`succ 3;`".

Le funzioni sono associative a sinistra, quindi l'espressione `f g x` è equivalente alla scrittura `(f g) x`.

Funzioni lineari implementate sulle liste

La maggior parte delle funzioni su liste operano su ogni elemento in una lista, ad esempio:

```
(*ritorna la lunghezza di una lista*)
fun length(x) = if null(x) then
                    0
                else
                    1 + lenght(tl(x));
```

Questa funzione è linearmente ricorsiva poiché si richiama solamente una volta sul lato destro.

Operazioni comuni sulle liste

La funzione `append(x,y)` ritorna una nuova lista con gli elemnti di `x` seguiti dagli elementi di `z`. Può essere definita nel seguente modo

```
fun append(x,z) = if null(x) then
                      z
                  else
                        hd(x)::append(tl(x),z);
```

Spiegazione L'algoritmo funziona esaurisce la lunghezza della lista `x` e ne costruisce una nuova tramite l'operatore `::` (quello che in Lisp viene chiamato `cons`). La prima condizione consiste nel caso base della ricorsione, mentre la seconda condizione è chiamata passo ricorsivo. Il suo funzionamento sarà fatto vedere in dettaglio più avanti. Siamo un esempio di esecuzione

```
append([1],[2]);
val it = [1, 2]: int list
```

La funzione `append` è definita all'interno del linguaggio come un operatore di concatenazione fra liste, indicato dal simbolo "`@`". Ad esempio

```
[1,2]@[3,4,5];
val it = [1,2,3,4,5] : int list
```

Funzionamento della funzione append

```
        append([1,2],[3,4,5])
=    1::append([2],[3,4,5])
= 1::2::append([],[3,4,5])
```

Qui si applica il caso base (in quanto la lista `x` è nulla), quindi di ritorna il parametro z

```
  1::2::[3,4,5]
= 1::[2,3,4,5]
= [1,2,3,4,5]
```

Studio della complessità Il costo dell'algoritmo dipende solamente dalla lunghezza del primo elemento, in questo caso una lista, di conseguenza ha complessità $\mathcal{O}(n)$.

La funzione `reverse(x,z)` ritorna una lista avente come testa la prima lista inversa e come coda la seconda lista. Ad esempio `reverse ([2,3,4],[1])` $\equiv$ `[4,3,2,1]`.

La funzione `reverse` è definita all'interno del linguaggio come la funzione `rev`. E possiamo dire che `rev(x)` $\equiv$ `reverse (x,[])`. La funzione `rev` è una funzione *wrapper* per `reverse (x,[])` contenente i due parametri x, la lista che gli abbiamo passato, e [], una lista di appoggio che diventerà la nostra nuova lista.

```
rev([1,2,3,4]); (* rev [1,2,3,4]; *)
val it = [4, 3, 2, 1]: int list
```

Se proviamo a identificare il caso base, è ovvio che `reverse([],z)` e `z` sono equivalenti, così come `reverse(a::y,z)` e `reverse (y,a::z)`. La funzione `reverse` è così definita:

```
fun reverse(x,z) = if null(x) then
                       z
                   else
                       reverse(tl(x),hd(x)::z);
```

A meno di definirla noi, non è possibile richiamre la funzione `reverse(x)`, l'interprete restituirà un errore nel caso provassimo ad invocarla.

Funzionamento della funzione reverse

```
  reverse([2,3,4],[1])
= reverse([3,4],2::[1])
= reverse([3,4],[2,1])
= reverse([4],3::[2,1])
= reverse([4],[3,2,1])
= reverse([], 4::[3,2,1])
= reverse([], [4,3,2,1]) (* caso base *)
= [4,3,2,1]
```

Complessità La funzione `reverse` è $\mathcal{O}(n)$ dove n è la lunghezza di `x`, ossia della lista in testa.